VPR and T-VPack[1] User's Manual

Summer 2008 VPR 5.0 Full Release, July 29, 2009

VPR Contributers:

Betz, Vaughn (vaughn@eecg.toronto.edu)
Campbell, Ted (campbell@eecg.toronto.edu)
Fang, Wei Mark (fang@eecg.toronto.edu)
Jamieson, Peter (jamieson@eecg.toronto.edu)
Kuon, Ian (ikuon@eecg.toronto.edu)
Luu, Jason (jluu@eecg.toronto.edu)
Marquardt, Alexander
Rose, Jonathon (jayar@eecg.toronto.edu)
Ye, Andy

1. Overview

VPR (Versatile Place and Route) is an FPGA placement and routing tool. VPR has four required and many optional parameters; it is invoked by typing:

```
> vpr netlist.net architecture.xml placement.p routing.r [-options]
```

Netlist.net is the netlist describing the circuit to be placed and/or routed, while architecture.xml describes the architecture of the FPGA in which the circuit is to be realized. If VPR is placing a circuit, the final placement will be written to placement.p; if VPR is routing a previously placed circuit, the placement is read from placement.p. The final routing of a circuit is written to file routing.r. The format of each of these files is described in Section 0.

VPR can be run in one of two basic modes. In its default mode, VPR places a circuit on an FPGA and then repeatedly attempts to route it in order to find the minimum number of tracks required by the specified FPGA architecture to route this circuit. If a routing is unsuccessful, VPR increases the number of tracks in each routing channel and tries again; if a routing is successful, VPR decreases the number of tracks before trying to route it again. Once the minimum number of tracks required to route the circuit is found, VPR exits. The other mode of VPR is invoked when a user specifies a specific channel width for routing. In this case, VPR places a circuit and attempts to route it only once, with the specified channel width. If the circuit will not route at the specified channel width, VPR simply report that it is unroutable.

VPR can perform either global routing or combined global and detailed routing.

T-VPack is a packing program which can be used with or without VPR. It takes a technology-mapped netlist (in blif format) consisting of lookup tables (LUTs), flip flops (FFs), and black boxes. It packs the LUTs and FFs together to form more coarse-grained logic blocks while treating inputs and outputs from black boxes as primary outputs and inputs respectively. The netlist it outputs is in the .net format required by VPR, and hence can be fed directly into VPR. Its usage is:

```
> t-vpack input.blif output.net [-options]
```

[1] *T-VPack is a timing-driven version of the VPack program that was provided with earlier versions of VPR. When run in its non-timing-driven mode, T-VPack is equivalent to VPack.*

Typing either VPR or T-VPack with no parameters will print out a list of all the available command line parameters.

2. Compiling VPR and T-VPack

If your compiler of choice is gcc and you are running a Solaris-based Sparcstation, you can compile VPR simply by typing *make* in the directory containing VPR's source code and makefile. If your compiler and/or architecture are different, however, you will have to make some small modifications to the makefile. First, change the CC = gcc line in the makefile so that CC is set to the name of your desired compiler. Second, you may want to change the line OPT_FLAGS = -O3 to set OPT_FLAGS to the value that gives the highest level of optimization with your compiler, and it may be necessary to give the linker different options so it finds all the relevant libraries on your machine. If, during compilation, you get an error that type XPointer is not defined, uncomment the "typedef char *XPointer" line in graphics.c (many X Windows implementations do not define the XPointer type). Finally, if you are compiling VPR on a system without X Windows (e.g. Windows NT), you should add a "#define NO_GRAPHICS" line to the top of vpr_types.h. VPR's built-in graphics will all be removed by this define, allowing compilation on non-X11 machines.

Project files are included for Microsoft Visual C++ 2005 for compilation under Windows. The Cygwin package should allow compiling under Windows using the makefile. Graphics are not supported when compiling with Visual C++, but should be supported when compiling with Cygwin.

If you are using T-VPack to convert SIS output to VPR's netlist format, you should make similar modifications to T-VPack's makefile.

3. Typical CAD Flow

Figure 1 illustrates the CAD flow we typically use. First, Odin [16] converts a Verilog Hardware Destription Language (HDL) design into a flattened netlist consisting of logic gates and blackboxes that represent heterogeneous blocks. Next, the ABC [1] synthesis package is used to perform technology-independent logic optimization of each circuit, and then each circuit is technology-mapped into 4-LUTs and flip flops [2]. The output of ABC is a .blif format netlist of LUTs, flip flops, and blackboxes. The T-VPack program [3, 4, 5, 6] then packs this netlist of LUTs and flip flops into more coarse-grained logic blocks, and outputs a netlist in the .net format VPR uses. The black boxes are also converted into the .net format. VPR [3, 4, 7, 8, 9, 10, 11] can then place the circuit and either globally route it or perform combined global and detailed routing on it. The output of VPR consists of a file describing the circuit placement, another file describing the circuit's routing, and various statistics concerning the minimum number of tracks per channel required to successfully route, the total wirelength, etc. In order to find the minimum number of tracks required for successful routing, VPR actually attempts to route the circuit several times with different numbers of tracks allowed per channel in each attempted routing.

Of course, many variations on this CAD flow are possible. It is possible to use other high-level synthesis tools to generate the blif files that are passed into ABC. Also, one can use different logic optimizers and technology mappers than ABC; just put the output netlist from your technology-mapper into .blif format and feed it into T-VPack. Alternatively, if the logic block you are interested in is not supported by T-VPack, your CAD flow can bypass T-VPack altogether by outputting a netlist of logic blocks in .net format. VPR can place and route netlists of any type of logic block -- you simply have to create the netlist and describe the logic block in the FPGA architecture description file. Finally, if you want only to route a placement produced by another CAD tool you can create a placement file in VPR format, and have VPR route this pre-existing placement.

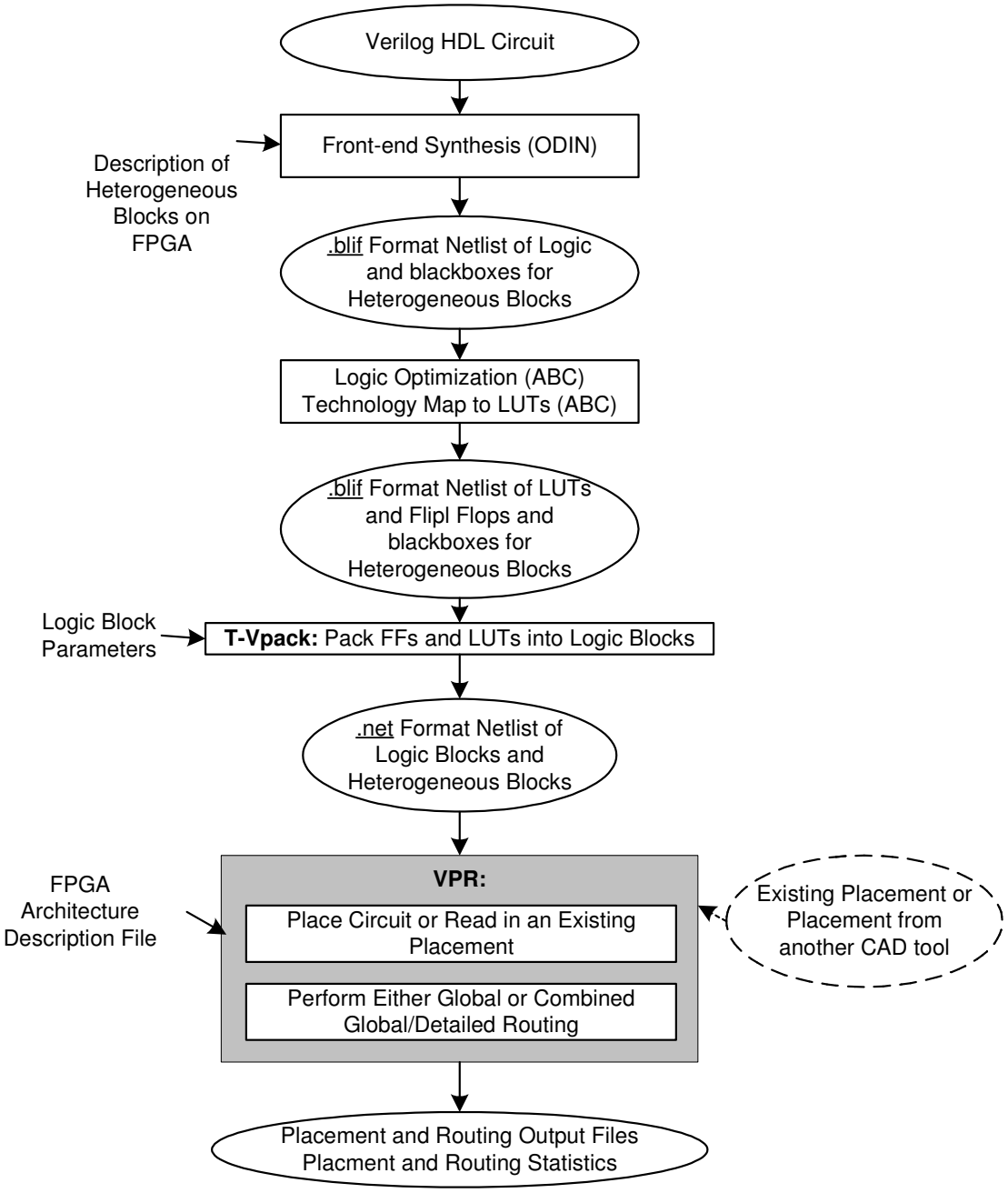

Verilog HDL Circuit
Front-end Synthesis (ODIN)
Description of Heterogeneous Blocks on FPGA
.blif Format Netlist of Logic and blackboxes for Heterogeneous Blocks
Logic Optimization (ABC) Technology Map to LUTs (ABC)
.blif Format Netlist of LUTs and Flipl Flops and blackboxes for Heterogeneous Blocks
Logic Block Parameters
T-Vpack: Pack FFs and LUTs into Logic Blocks
.net Format Netlist of Logic Blocks and Heterogeneous Blocks
VPR:
FPGA Architecture Description File
Place Circuit or Read in an Existing Placement
Perform Either Global or Combined Global/Detailed Routing
Existing Placement or Placement from another CAD tool
Placement and Routing Output Files
Placment and Routing Statistics

Figure 1

4. Operation of T-VPack

As stated earlier, T-VPack takes as input a technology-mapped netlist of lookup tables (LUTs) and flip flops in .blif format, and outputs a .net format netlist composed of more complex logic blocks. The logic block to be targeted is selected via command-line options. The simplest logic block T-VPack can target consists of a LUT and a FF, in the configuration shown in Figure 2. We call this logic block a basic logic element.

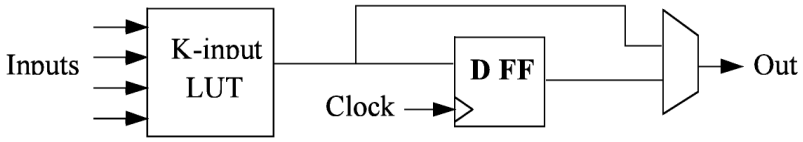


Figure 2: Basic logic element.

To have T-VPack target a logic block of this form, use the command:

```
> t-vpack <input.blif> <output.net> -lut_size <K> -no_clustering
```

In the command above, the italicized values in angled brackets, <>, should be replaced by the file names or numbers you are using, while unitalicized words are keywords and must be typed exactly as shown.

The **-lut_size <K>** option specifies the number of inputs to a LUT (i.e. K in Figure). If -lut_size is not specified, a default LUT size of 4 is assumed by T-VPack. The **-no_clustering** option indicates that the logic block is a single basic logic element with no local routing to route the logic block output back to the logic block inputs. By default, T-VPack marks all clock nets in the input netlist as global nets which VPR should not route. Since clocks are typically routed via a dedicated network in FPGAs, this is usually the most realistic thing to do. If, however, you want clocks to be routed as using normal routing resources, you should specify **-global_clocks off** on the T-VPack command line.

T-VPack is capable of targeting a more complex form of logic block, which we call a cluster-based logic block [5]. Figure 3 depicts an example. A cluster-based logic block consists of N basic logic elements (i.e. N LUTs and N FFs), along with local interconnect that allows the N cluster outputs to be routed back to LUT inputs. Since the number of logic block inputs, I, can be less than the total number of LUT inputs (KN, where K is the number of inputs per LUT), the local interconnect also allows each of the I inputs to be routed to any of the KN LUT inputs. Cluster-based logic blocks are very similar to the logic blocks used in the Altera 8K and 10K FPGAs, and are reasonably similar to those used in the Xilinx 5200 and Virtex FPGAs.

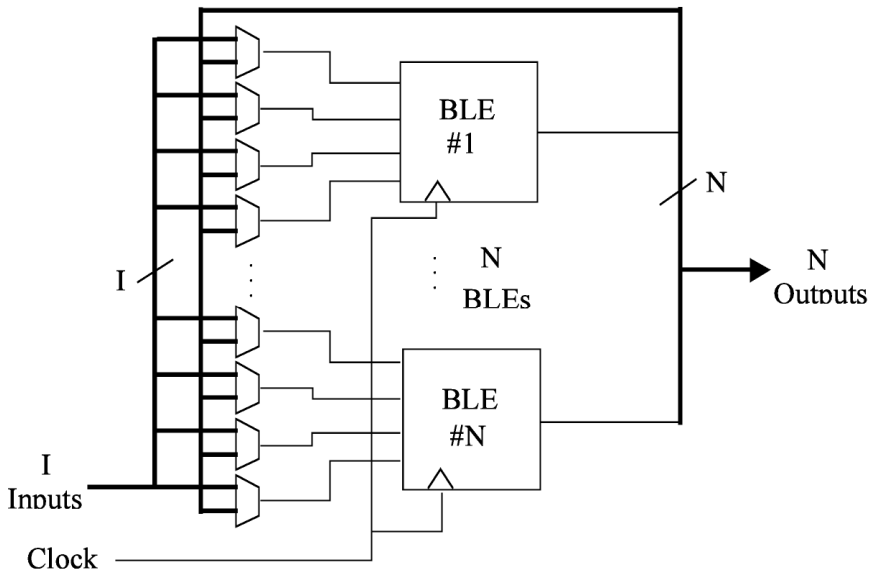


Figure 3: A cluster-based logic block.

To target such a logic block, use a command line of the form:

```
> t-vpack input.blif output.net -lut_size <K> -cluster_size <N>
  -inputs_per_cluster <I> -clocks_per_cluster <C>
```

The meaning of the -inputs_per_cluster and -cluster_size parameters should be clear from Figure . The -clocks_per_cluster option is used to specify how many distinct clocks can be used by each logic block.

4.1 T-VPack Options

4.1.1 Architecture Description Options That Are Always Valid

-lut_size <int>
Number of inputs per LUT (i.e. *K*). Default: 4.
-no_clustering
Specifies that no clustering is to be performed -- i.e. the logic block consists of one BLE (a LUT and a FF) with no local routing. Default: cluster.
-global_clocks {on | off}
Indicates whether clocks should be marked as being routed via a special, global resource. VPR does not route global signals. Default: on.

4.1.2 Architecture Options Valid Only When -no_clustering Is Not Specified

-cluster_size <int>
Number of BLEs in a cluster-based logic block (i.e. N).

Default: 1.

-inputs_per_cluster <int>

Number of distinct inputs in a logic cluster (i.e. *I*).

*Default: lut_size * cluster_size.*

-clocks_per_cluster <int>

Number of distinct clocks in a logic cluster.

Default: 1.

-muxes_to_cluster_output_pins {on | off}

If "off", each BLE output is hooked directly to a cluster output pin. If "on", a set of *N* (one per cluster output) *N*:1 multiplexers allows each output pin to be driven by any of the *N* BLEs within a cluster.

Default: off.

4.1.3 CAD Optimization Options

-timing_driven {on | off}

Controls whether the clustering algorithm attempts to optimize circuit timing by attempting to capture critical connections within a logic cluster.

Default: on.

-connection_driven {on | off}

Controls whether or not T-VPack attempts to absorb, within one cluster, connections from the output of one BLE to the input of another.

Default: off.

-hill_climbing {on | off}

Controls whether the algorithm used to pack BLEs into clusters allows hill climbing or is strictly greedy.

Default: on.

-cluster_seed {timing | max_inputs}

Specifies the way in which the cluster packing algorithm picks the first BLE to be placed in an empty cluster. Max_inputs picks the BLE with the most used inputs, while timing picks the BLE on the most critical path.

Default: timing if timing_driven is on, max_inputs otherwise.

-allow_unrelated_clustering {on | off}

Controls whether or not BLEs with no attraction to the current cluster can be packed into it.

Default: on.

-alpha <float>

A tradeoff parameter that controls the optimization of delay in packing vs. the optimization of signal sharing. A value of 0 focuses solely on signal sharing, while a value of 1 focuses solely on timing. This option is meaningful only when timing_driven is on.

Default: 0.75.

-recompute_timing_after <int>

T-VPack will recompute its estimate of how timing-critical each connection is after packing the specified number of BLEs into clusters. This option is meaningful only when timing_driven is on.

Default: 32 000.

-block_delay <float>

The relative delay of a BLE. This option is meaningful only when timing_driven is on.

Default: 0.1.

-intra_cluster_net_delay <float>

The relative delay of a signal that goes from one BLE to another using the local routing within a cluster. This option is meaningful only when timing_driven is on.

Default: 0.1.

-inter_cluster_net_delay <float>

The relative delay of a signal that goes from one BLE to another BLE that is in a different cluster, or an IO pad. This option is meaningful only when timing_driven is on.

Default: 1.0.

-allow_early_exit {on | off}

If on, the clusterer will stop re-timing analyzing a circuit once it believes the current, partially complete packing, has fixed ("locked") the critical path.

Default off.

5. Operation of VPR

Invoke VPR by typing:

```
> vpr input.net input.arch placement.p output.routing [-options]
```

This section outlines how VPR's graphics and options work; Section describes the format of each of the four files used by VPR.

5.1 Graphics

The graphics included in VPR are very easy to use. Click any mouse button on the **arrow** keys to pan the view, or click on the **Zoom-In**, **Zoom-Out** and **Zoom-Fit** keys to zoom the view. Click on the **Window** button, then on the diagonally opposite corners of a box, to zoom in on a particular area. Selecting **PostScript** creates a PostScript file (in pic1.ps, pic2.ps, etc.) of the image on screen. **Proceed** tells VPR to continue with the next step in placing and routing the circuit, while **Exit** aborts the program. The menu buttons will be greyed out to show they are not selectable when VPR is working, rather than interactively displaying graphics.

The **Toggle Nets** button toggles the nets in the circuit visible/invisible. When a placement is being displayed, routing information is not yet known so nets are simply drawn as a "star;" that is, a straight line is drawn from the net source to each of its sinks. Click on any clb in the display, and it will be highlighted in green, while its fanin and fanout are highlighted in blue and red, respectively. Once a circuit has been routed the true path of each net will be shown. Again, you can click on Toggle Nets to make net routings visible or invisible, and clicking on a clb or pad will highlight their fanins and fanouts.

When a routing is on-screen, clicking on **Toggle RR** will switch between various views of the routing resources available in the FPGA. Wiring segments and clb pins are drawn in black, connections from wiring segments to input pins are shown in blue, connections from output pins to wiring segments are shown in red, and connections between wiring segments are shown in green. The points at which wiring segments connect to clb pins (connection box switches) are marked with an "X". Switch box connections will have buffers (triangles) or pass transistors (circles) drawn on top of them, depending on the type of switch each connection uses. Clicking on a clb or pad will overlay the routing of all nets connected to that block on top of the drawing of the FPGA routing resources, and will label each of the pins on that block with its pin number. The routing resource view can be very useful in ensuring that you have correctly described your FPGA in the architecture description file -- if you see switches where they shouldn't be or pins on the wrong side of a clb, your architecture description needs to be revised.

When a routing is shown on-screen, clicking on the **Congestion** button will show any overused routing resources (wires or pins) in red, if any overused resources exist. Finally, when a routing is on screen you can click on the **Crit. Path** button to see each of the nets on the critical path in turn. The current net on the critical path is highlighted in cyan; its source block is shown in yellow and the critical sink is shown in green.

NOTE: For this release, a few of the less common options are not fully tested and so are not necessarily working properly. As well, the –nx –ny and –aspect_ratio options are now part of the architecture file. The options not debugged are:

- *num_regions*
- *base_cost_type*
- *place_cost_type*

5.2 Command-Line Options

VPR has a lot of options. The options most people will be interested in are **-inner_num, -route_chan_width,** and **-route_type**. In general for the other options the defaults are fine, and only people looking at how different CAD algorithms perform will try many of them. To understand what the more esoteric placer and router options actually do, buy [3] or download [7, 8, 9, 10] from the author's web page (http://www.eecg.toronto.edu/~vaughn).

In the following text, values in angle brackets, e.g. <int>, should be replaced by the appropriate filename or number. Values in curly braces separated by vertical bars, e.g. {on | off}, indicate all the permissible choices for an option.

5.2.1 General Options

-nodisp
Disables all graphics. Useful if you're not running X Windows. *Default: graphics enabled.*
-auto <int>
Can be 0, 1, or 2. This sets how often you must click Proceed to continue execution after viewing the graphics. The higher the number, the more infrequently the program will pause. *Default: 1.*
-route_only
Take an existing placement from the placement file specified on the command line and route it. *Default: off.*
-place_only
Place the circuit, but do not route it. *Default: off.*
-timing_analysis { on | off }
Turn timing analysis of the routing on or off. If it is off, you don't have to specify the various timing analysis parameters in the architecture file. *Default: on, unless architecture file does not have timing information*
-timing_analyze_only_with_net_delay <float>
Perform timing analysis on netlist assuming all edges have the same specified delay *Default: off*
-outfile_prefix <string>
Prefix output files with specified string.
-full_stats
Print out some extra statistics about the circuit and its routing useful for wireability analysis. *Default: off*

5.2.2 Placer Options

By default, the automatic annealing schedule [3, 9] is used. This schedule gathers statistics as the placement progresses, and uses them to determine how to update the temperature, when to exit, etc. This schedule is generally superior to any user-specified schedule. If any of init_t, exit_t or alpha_t is specified, the user schedule, with a fixed initial temperature, final temperature and temperature update factor is used.

-seed <int>

Sets the initial random seed used by the placer.

Default: 1.

-num_regions <int>

Used only with the nonlinear cost function. VPR will compute congestion on an array of num_regions X num_regions subareas. Large values of num_regions greatly slow the placer.

Default: 4.

Note: This is not supported and may not be working this release

-enable_timing_computations {on | off}

Controls whether or not the placement algorithm prints estimates of the circuit speed of the placement it generates. This setting affects statistics output only, not optimization behaviour.

Default: on if timing-driven placement is specified, off otherwise.

-block_dist <int>

Specifies that the placement algorithm should print out an estimate of the circuit critical path, assuming that each inter-block connection is between blocks a (horizontal) distance of block_dist logic blocks apart. This setting affects statistics output only, not optimization behaviour.

Default: 1. (Currently the code that prints out this lower bound is #ifdef 'ed out in place.c -- define PRINT_LOWER_BOUND in place.c to reactivate it.)

-inner_num <float>

The number of moves attempted at each temperature is inner_num * num_blocks^(4/3) in the circuit. The number of blocks in a circuit is the number of pads plus the number of clbs. Changing inner_num is the best way to change the speed/quality tradeoff of the placer, as it leaves the highly-efficient automatic annealing schedule on and simply changes the number of moves per temperature.

Note: Specifying -inner_num 1 will speed up the placer by a factor of 10 while typically reducing placement quality only by 10% or less (depends on the architecture). Hence users more concerned with CPU time than quality may find this a more appropriate value of inner_num.

Default: 10.

-init_t <float>

The starting temperature of the anneal for the manual annealing schedule.

Default: 100.

-exit_t <float>

The (manual) anneal will terminate when the temperature drops below the exit temperature.

Default: 0.01.

-alpha_t <float>

The temperature is updated by multiplying the old temperature by alpha_t when the manual annealing schedule is enabled.

Default: 0.8.

-fix_pins {random | <file.pads>}

Do not allow the placer to move the I/O locations about during the anneal. Instead, lock each I/O pad to some location at the start of the anneal.

If **-fix_pins random** is specified, each I/O block is locked to a random pad location to model the effect of poor board-level I/O constraints. If any word other than random is specified after -fix_pins, that string is taken to be the name of a file listing the desired location of each I/O block in the netlist (i.e. **-fix_pins <file.pads>**).

This pad location file is in the same format as a normal placement file, but only specifies the locations of I/O pads, rather than the locations of all blocks.

Default: off (i.e. placer chooses pad locations).

-place_algorithm {bounding_box | net_timing_driven | path_timing_driven}

Controls the algorithm used by the placer.

Bounding_box focuses purely on minimizing the bounding box wirelength of the circuit.

Path_timing_driven focuses on minimizing both wirelength and the critical path delay.

Net_timing_driven is similar to path_timing_driven, but assumes that all nets have the same delay when estimating the critical path during placement, rather than using the current placement to obtain delay estimates.

Default: path_timing_driven.

-place_cost_type {linear | nonlinear}

Select the (wirelength portion of the) placement cost function. For FPGAs in which all channels have the same width the linear cost function reduces to a bounding box wirelength cost function. The nonlinear cost function, on the other hand, considers both wirelength and congestion during placement.

Default: linear.

Note: Nonlinear is not supported this release and may give unusual results

-place_chan_width <int>

Can be used with the nonlinear cost function to tell VPR how many tracks a channel of relative width 1 is expected to need to complete routing of this circuit. VPR will then place the circuit only once, and repeatedly try routing the circuit as usual. If place_chan_width is not specified and the nonlinear cost is used, VPR will replace and reroute the circuit for each channel width at which it attempts to map the circuit.

5.2.3 Placement Options Valid Only With Timing-Driven Placement

Timing Driven placement is used by default, unless the architecture file is missing timing information.

-timing_tradeoff <float>

Controls the trade-off between bounding box minimization and delay minimization in the placer. A value of 0 makes the placer focus completely on bounding box (wirelength) minimization, while a value of 1 makes the placer focus completely on timing optimization.

Default: 0.5.

-recompute_crit_iter <int>

Controls how many temperature updates occur before the placer performs a timing analysis to update its estimate of the criticality of each connection.

Default: 1.

-inner_loop_recompute_divider <int>

Controls how many times the placer performs a timing analysis to update its criticality estimates while at a single temperature.
Default: 0.

-td_place_exp_first <float>

Controls how critical a connection is considered as a function of its slack, at the start of the anneal. If this value is 0, all connections are considered equally critical. If this value is large, connections with small slacks are considered much more critical than connections with small slacks. As the anneal progresses, the exponent used in the criticality computation gradually changes from its starting value of *td_place_exp_first* to its final value of *td_place_exp_last*.
Default: 1.

-td_place_exp_last <float>

Controls how critical a connection is considered as a function of its slack, at the end of the anneal.

See discussion for *-td_place_exp_first*, above. *Default: 8.*

5.2.4 Router Options

-max_router_iterations <int>
The number of iterations of a Pathfinder-based router that will be executed before a circuit is declared unrouteable (if it hasn't routed successfully yet) at a given channel width. *Default: 50.* *Speed-quality trade-off: reduce this number to speed up the router, at the cost of some increase in final track count. This is most effective if -initial_pres_fac is simultaneously increased.*
-initial_pres_fac <float>
Sets the starting value of the present overuse penalty factor. *Default: 0.5.* *Speed-quality trade-off: increase this number to speed up the router, at the cost of some increase in final track count. Values of 1000 or so are perfectly reasonable.*
-first_iter_pres_fac <float>
Similar to -initial_pres_fac. This sets the present overuse penalty factor for the very first routing iteration. -initial_pres_fac sets it for the second iteration. *Default: 0.5.*
-pres_fac_mult <float>
Sets the growth factor by which the present overuse penalty factor is multiplied after each router iteration. *Default: 1.3.*
-acc_fac <float>
Specifies the accumulated overuse factor (historical congestion cost factor). *Default: 1.*
-bb_factor <int>
Sets the distance (in channels) outside of the bounding box of its pins a route can go. Larger numbers slow the router somewhat, but allow for a more exhaustive search of possible routes. *Default: 3.*
-base_cost_type {demand_only | delay_normalized | intrinsic_delay}
Sets the basic cost of using a routing node (resource). Demand_only sets the basic cost of a node according to how much demand is expected for that type of node. Delay_normalized is similar, but normalizes all these basic costs to be of the same magnitude as the typical delay through a routing resource. Intrinsic_delay sets the basic cost of a node to its intrinsic delay. *Default: delay_normalized for the timing-driven router and demand_only for the breadth-first router.* *Note: intrinsic_delay is not supported this release and may give unusual results*
-bend_cost <float>
The cost of a bend. Larger numbers will lead to routes with fewer bends, at the cost of some increase in track count. If only global routing is being performed, routes with fewer bends will be easier for a detailed router to subsequently route onto a segmented routing architecture. *Default: 1 if global routing is being performed, 0 if combined global/detailed routing is being performed.*
-route_type {global | detailed}
Specifies whether global routing or combined global and detailed routing should be performed.

Default: detailed (i.e. combined global and detailed routing).

-route_chan_width <int>

Tells VPR to route the circuit with a certain channel width. No binary search on channel capacity will be performed to find the minimum number of tracks required for routing -- VPR simply reports whether or not the circuit will route at this channel width.

-router_algorithm {breadth_first | timing_driven | directed_search}

Selects which router algorithm to use. The breadth-first router focuses solely on routing a design successfully, while the timing-driven router focuses both on achieving a successful route and achieving good circuit speed. The breadth-first router is capable of routing a design using slightly fewer tracks than the timing-driving router (typically 5% if the timing- driven router uses its default parameters; this can be reduced to about 2% if the router parameters are set so the timing-driven router pays more attention to routability and less to area). The designs produced by the timing-driven router are much faster, however, (2x - 10x) and it uses less CPU time to route. The directed_search router is routability-driven and uses an A* heuristic to improve runtime over breadth_first.

Default: timing_driven.

5.2.5 Timing-Driven Router Options

-astar_fac <float>
Sets how aggressive the directed search used by the timing-driven router is. Values between 1 and 2 are reasonable, with higher values trading some quality for reduced CPU time. *Default: 1.2.*
-max_criticality <float>
Sets the maximum fraction of routing cost that can come from delay (vs. coming from routability) for any net. A value of 0 means no attention is paid to delay; a value of 1 means nets on the critical path pay no attention to congestion. *Default: 0.99.*
-criticality_exp <float>
Controls the delay - routability tradeoff for nets as a function of their slack. If this value is 0, all nets are treated the same, regardless of their slack. If it is very large, only nets on the critical path will be routed with attention paid to delay. Other values produce more moderate tradeoffs. Default: 1.

6. File Formats

In all the file format that follow, a sharp (#) character anywhere in a line indicates that the rest of the line is a comment, while a backslash (\) at the end of a line (and not in a comment) means that this line is continued on the line below.

6.1 Circuit Netlist (.net) Format

Three different circuit elements are available: input pads, output pads, and functional blocks. Input and output pads are specified using the keywords .input and .output while functional blocks are specified by .[name], respectively. The .[name] for the functional block must correspond with the .[name] specified in the architecture file. For example, .clb in the netlist is specified by a .clb in the architecture file. The format is shown below.

```
element_type_keyword  blockname
   pinlist: net_a  net_b  net_c ...
   subblock: subblock_name pin_num1 pin_num2 ...  # Only needed if a
functional block
```

A circuit element is created by specifying a keyword at the start of a line, followed by the name to be used to identify this block. The line immediately below this keyword line starts with the identifier *pinlist:* and then lists the names of the nets connected to each pin of the functional block or pad. Input and output pads (*.inputs* and *.outputs*) have only one pin, while functional blocks (*.[name]*) have as many pins as the architecture file used for this run of VPR specifies. The first net listed in the pinlist connects to pin 0 of a functional block, and so on. If some pin of a functional block is to be left unconnected, the corresponding entry in the pinlist should specify the reserved word *open* instead of a net name.

Functional blocks (.[name]) also have to specify the internal contents of the functional block with *subblock* lines. Each functional block must have at least one subblock line, and can have up to *max_subblocks* attribute, where *max_subblocks* is set in the architecture file. A functional block may have less than *max_subblocks* subblock lines, since some of the subblocks in the functional block may be unused. Each subblock is a K-input O-output boolean logic element (BLE) (where K is set via the *max_subblock_inputs* attribute and O is set via the

max_subblock_outputs attribute in the architecture description file) and a flip flop, as shown in Figure . The subblock line first gives the name of the subblock, and then gives the functional block pin or a subblock output pin within this functional block to which each BLE pin is connected. If a BLE pin is unconnected, the corresponding pin entry should be set to the keyword open. The order of the BLE pins is: *max_subblock_inputs* input pins, *max_subblock_outputs* output pins, and the clock input (*max_subblock_inputs* + *max_subblock_outputs* + 1 pins total).

Each of the subblock BLE input pins can be connected to any of the functional block input pins, or to the output of any of the subblocks in this functional block. A connection to a functional block input pin is specified by giving the number of the functional block pin in the appropriate place, while a connection to a subblock output is specified by "ble_<subblock_number>". For example, to connect to functional block pin 0, one lists 0 in the appropriate place, while to connect to the output of subblock 0, one lists ble_0 in the appropriate place. Each subblock clock pin can similarly be connected to either a clb input pin or the output of a subblock in the same logic block. If the subblock clock pin is "open" all the BLE outputs are unregistered outputs; otherwise all the BLE output are assumed to be registered. The entry corresponding to the subblock output pin specifies the number of the functional block output pin to which it connects, or *open* if this subblock output is doesn't connect to any clb output pin (which happens when a subblock output is used only locally, within a logic block).

The only other keyword is *.global*. Use .global lines to specify that a net or nets should not be considered by the placement cost function or routed. It is assumed that some global routing resources exist to route these very high fanout signals (generally clocks). The syntax of the .global statement is:

```
.global net_a net_b ...
```

An example netlist in which the logic block is a single BLE is given below.

```
#This netlist describes a small circuit with two inputs
#and one output. There is only one clb block, which is
#a 3-input BLE (LUT+FF) that has one unconnected input.
#This netlist assumes that the architecture input file defines
#a clb as a 3-input BLE with pins 0, 1, and 2 being the LUT inputs,
#pin 3 being the LUT output, and pin 4 being the BLE clock.

.input a                        #Input pad.
      pinlist: a                #Blocks can have the same
                                #name as nets with no conflict.

.input bpad
      pinlist: b

.clb simple                     # Logic block.
      pinlist: a b open and2 open          # 2 LUT inputs used,
                                           # clock input unconnected.
      subblock: sb_one 0 1 open 3 open     # Subblock line says the
                                           # same thing.

.output out_and2                    #Output pad.
      pinlist: and2
```

In the netlist above the subblock line adds no new information -- since the logic block only contains one BLE, which pins are hooked to this BLE is obvious. Consider a netlist in which

each logic block is a cluster-based logic block containing two subblocks, or BLEs, and let there also be a multiplier block:

```
.input a
      pinlist: a

.input bpad
      pinlist: b

.input c
      pinlist: c

.input d
      pinlist: d

.input clk
      pinlist: clk

.global clk       # Typical case:  clock needn't be routed, as there's a
                  # special network for it.

# Example logic block:  4 inputs, 2 outputs, 1 clock.
# Internally, the logic block contains two BLEs,
# each of which consists of a 3-LUT and a FF.
# Local routing allows subblock outputs to connect to subblock inputs
# in the same logic block.

.clb more_complex
      pinlist:  a b c open out_1 out_2 clk
      subblock: sb_zero 0 1 open 4 open   # BLE inputs are a and b, output
                                          # goes to out_1. Output isn't
                                          # registered.
      subblock: sb_one ble_0 1 2 5 6      # BLE inputs are the output of
                                          # subblock 0,
                                        # and nets b and c. The output
                                        # goes to out_2.
                                        # The output is registered.
.mult two_by_two
      pinlist:  a b c d out_3 out_4 out_5 out_6 open # Combinational 2x2
                                                     # multiply with 4
                                                     #inputs and 4 outputs
      subblock: sb_zero 0 1 2 3 4 5 open   # BLE inputs are a, b, c, d
                                           #outputs are out_3, out_4, out_5,
                                           # and out_6.  Outputs are not
                                         # registered.

.output opad_1
      pinlist: out_1

.output opad_2
      pinlist: out_2

.output opad_3
      pinlist: out_3

.output opad_4
      pinlist: out_4

.output opad_5
      pinlist: out_5
```

```
.output opad_6
      pinlist: out_6
```

In the netlist above, one needs the subblock statements to know what connections are made internally to the logic block by local routing. Figure 4 shows the connections this netlist describes for the clb "more_complex" only. Note also that while the subblock lines describe the internal structure of the clb in terms of BLEs, the BLE structure is general enough that the timing behaviour of essentially arbitrary logic blocks can be described in terms of subblock lines. VPR needs the subblock information in a netlist only for timing analysis.

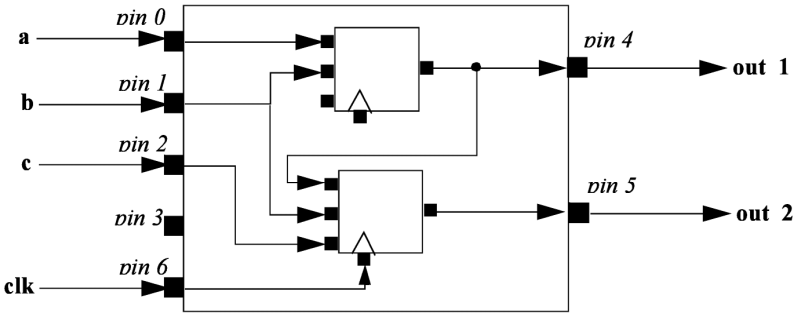


Figure 4: Connections within logic block "more_complex" specified by subblock lines of netlist above.

6.2 FPGA Architecture File (.xml) Format

The architecture file is specified in xml format. It is composed of a hierarchy of start and end tags with optional attributes and content inside each tag giving additional information. As a convention, curly brackets {…} represents an option with each option separated by |. For example, `a={1 | 2 | open}` means field "a" can take a value of 1, 2, or open.

The first tag in all architecture files is the `<architecture>` tag. This tag contains all other tags in the architecture file. The architecture tag contains five other tags. They are `<layout>` `<device>`, `<switchlist>`, `<segmentlist>`, and `<typelist>`.

<layout {auto="float" | **width=**"*int*" **height=**" *int*"**}/>**
This tag specifies the size and shape of the FPGA in grid units. The keyword *auto* indicates that the size should be chosen to be the minimal dimensions that fits the given circuit. The size is determined from the number grid tiles used by the circuit as well as the number of IO pins that it uses. The aspect ratio of the FPGA is given after the *auto* keyword and is the ratio width/height. Alternately, the size can be explicitly given as the size in the x direction (width) followed by the size in the y direction (height).
<device>content**</device>**
Content inside this tag specifies device information. It contains the tags <sizing>, <timing>, <area>, <chan_width_distr>, and <switch_block>.
<switchlist>content**</switchlist>**
Content inside this tag contains a group of <switch> tags that specify the types of switches and their properties.
<segmentlist>content**</segmentlist>**
Content inside this tag contains a group of tags that specify the types of wire segments and their properties.

< typelist>content**</ typelist>**
Content inside this tag contains a group of <type> tags that specify the types of functional blocks and their properties.

6.2.1 Description of Device Information in the FPGA

The tags within the device tag are described in the following table.

<sizing R_minW_nmos="*float***" R_minW_pmos="***float***" ipin_mux_trans_size="***int***"/>**
Specifies parameters used by the area model built into VPR R_minW_nmos attribute: The resistance of minimum-width nmos transistor. This data is used only by the area model built into VPR. R_minW_pmos attribute: The resistance of minimum-width pmos transistor. This data is used only by the area model built into VPR. ipin_mux_trans_size attribute: This specifies the size of each transistor in the ipin muxes. Given in minimum transistor units. The mux is implemented as a two-level mux.).
<timing C_ipin_cblock="*float***" T_ipin_cblock="***float***"/>**
Optional. Attributes specify timing information general to the device and must be specified for timing analysis. C_ipin_cblock attribute: Input capacitance of the buffer isolating a routing track from the connection boxes (multiplexers) that select the signal to be connected to an logic block input pin. One of these buffers is inserted in the FPGA for each track at each location at which it connects to a connection box. For example, a routing segment that spans three logic blocks, and connects to logic blocks at two of these three possible locations would have two isolation buffers attached to it. If a routing track connects to the logic blocks both above and below it at some point, only one isolation buffer is inserted at that point. If your connection from routing track to connection block does not include a buffer, set this parameter to the capacitive loading a track would see at each point where it connects to a logic block or blocks. T_ipin_cblock attribute: Delay to go from a routing track, through the isolation buffer (if your architecture contains these) and a connection block (typically a multiplexer) to a logic block input pin.
<area grid_logic_tile_area="*float***"/>**

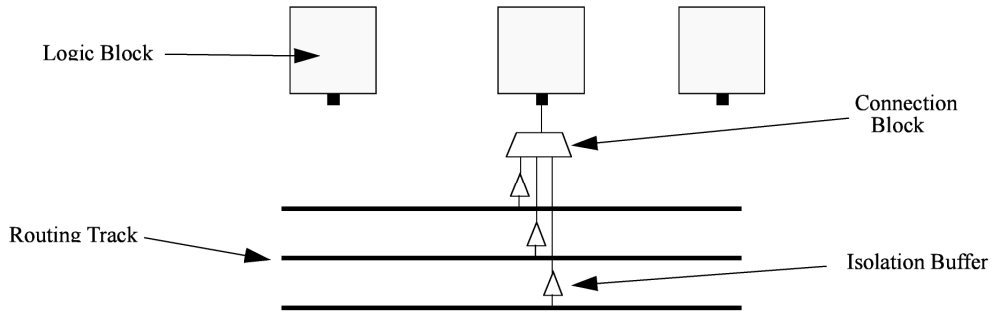


Figure 8: Routing track to logic block connection structure.

Used for an area estimate of the amount of area taken by all the functional blocks.
<switch_block type="{wilton|subset|universal}" fs="*int***"/>**
C When using bidirectional segments, all the switch blocks [12] have $F_s = 3$. That is, whenever horizontal and vertical channels intersect, each wire segment can connect to three other wire segments. The exact topology of which wire segment connects to which can be one of three choices. The *subset* switch box is the planar or domain-based switch box used in the Xilinx 4000 FPGAs -- a wire segment in track 0 can only connect to other wire segments in track 0 and so on. The *wilton* switch box is described in [13], while the *universal* switch box is described in [14]. To see the topology of a switch box, simply hit the "Toggle RR" button when a completed routing is on screen in VPR. In general the wilton switch box is the best of these three topologies and leads to the most routable FPGAs. When using unidirectional segments, a modified *wilton* switch block pattern is used regardless of the specified switch_block_type.
<chan_width_distr>content**</chan_width_distr>**
Content inside this tag is described in the next table

If global routing is to be performed, channels in different directions and in different parts of the FPGA can be set to different relative widths. This is specified in the content within the <chan_width_distr> tag. *If detailed routing is to be performed, however, all the channels in the FPGA must have the same width.*

<io width= "*float***"/>**
Width of the channels between the pads and core relative to the widest core channel.
<x distr="{gaussian|uniform|pulse|delta}" peak="*float***" width="** *float***" xpeak="** *float***" dc="** *float***"/>**
(*Unknown if works properly)* The italicized quantities are needed only for pulse, gaussian, and delta (which doesn't need width). Most values are from 0 to 1. Sets the distribution of tracks for the x-directed channels -- the channels that run horizontally. If uniform is specified, you simply specify one argument, peak. This value (by convention between 0 and 1) sets the width of the x-directed core channels relative to the y-directed channels and the channels between the pads and core. Figure should make the specification of uniform (dashed line) and pulse (solid line) channel widths more clear. The gaussian keyword takes the same four parameters as the pulse keyword, and they are all interpreted in exactly the same manner except that in the gaussian case width is the standard deviation of the function.

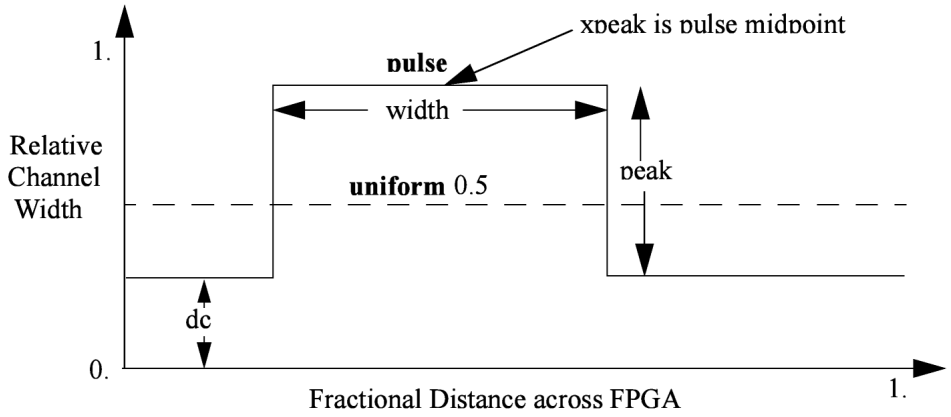


Figure 5: Specification of relative channel widths.

The delta function is used to specify a channel width distribution in which all the channels have the same width except one. The syntax is chan_width_x delta peak xpeak dc. Peak is the extra width of the single wide channel. Xpeak is between 0 and 1 and specifies the location within the FPGA of the extra-wide channel -- it is the fractional distance across the FPGA at which this extra-wide channel lies. Finally, dc specifies the width of all the other channels. For example, the statement *chan_width_x delta 3 0.5 1* specifies that the horizontal channel in the middle of the FPGA is four times as wide as the other channels.

Examples:

<x distr=”uniform” peak=”1”/>

<x distr=”gaussian” width=”0.5” peak=”0.8” xpeak=”0.6” dc=”0.2”/>

<y distr=”{gaussian|uniform|pulse|delta}” peak=” *float*” **width=”** *float*” **xpeak=”** *float*” **dc=”** *float*”**/>**

Sets the distribution of tracks for the y-directed channels.

6.2.2 Description of Functional Blocks in FPGA

The content within the `<typelist>` tag consists of a group of `<type>` tags. Each `<type name="<string>" height="<int>">` tag describes a functional block. The name attribute is the name for the functional block and correspond exactly with the name for the block in the netlist. It is of the format .[name]; for example, .clb. The height attribute specifies how many grid tiles the functional block takes up. The `<type>` tag contains tags specified in the following table.

<fc_in type="{frac|abs|full}">{<int> | <float>}</fc_in>

Content:

Sets the number of tracks to which each logic block input pin connects in each channel bordering the pin. The F_c value used is always the minimum of the specified F_c and the channel width, W. It is best to set the type attribute to *full* if you want F_c to always be W.

type attribute:

The type attribute indicates whether the F_c [12] value should be interpreted as the number of tracks to which each pin connects (*absolute*), or the fraction of tracks in a channel to which each pin connects (*fractional)*. Note: type absolute or fractional for Fc_in and Fc_out must be the same. Full disregards whether it is fractional or absolute and can be used in either case.

<fc_out type="{frac|abs|full}">{<int> | <float>}</fc_out>

Content:

Sets the number of tracks to which each logic block output pin connects in each channel bordering the pin.

Type attribute is the same as described above in Fc_in

<pinclasses>

Contains a group of pin classes in the form of `<class>` tags. A `<class>` tag is defined as:

```
<class type="{in|out|global}">(<int> )*</class>
```

Where * represents a Kleene star and brackets for regular expression grouping only (Do not put brackets in the architecture file). The type attribute specifies if pin numbers specified for this class are inputs, outputs, or global. All pins with the same class number are logically equivalent -- such as all the inputs of a LUT. Class numbers must start at zero and be consecutive. The global keyword is optional; if specified, it comes after the class number. Global input pins can connect only to signals marked as global in the netlist (typically clocks). Global input pins are not connected into the normal routing; it is assumed they connect to a special, dedicate resource used for special nets like clocks.

NOTE: The order in which your inpin and outpin statements appear must be the same as the order in which your netlist (.net) file lists the connections to the clbs. For example, if the first pin on each clb in the netlist file is the clock pin, your first pin statement in the architecture file must be an inpin statement defining the clock pin.

Pads are always assumed to have only one pin (either an input or an output), and this pin is accessible from the one channel bordering that pad. Hence no inpin or outpin statements are given for pads.

Declares an input pin, determines the class to which this pin belongs, and sets the side(s) of CLBs on which the physical output pin connection(s) is (are).

<pinlocations>

Contains a group of pin locations in the form of `<loc>` tags. A `<loc>` tag is defined as:

```
<loc side="{left|right|bottom|top}" offset=”<int>”><int>*</class>
```

Where * represents a Kleene star and brackets for regular expression grouping only (Do not put brackets in the architecture file). The side attribute specifies which of the four directions the pins in the contents are located on and offset attribute specifies the grid distance from the bottom grid tile that the pin is specified for. Pins on the bottom grid tile does not have an offset attribute. The offset value must be less than the height of the functional block. A functional block may not contain pins inside of itself.

Physical equivalence is specified by listing a pin number more than once for different locations.

<gridlocations>

Specifies the columns on the FPGA that will consist of this functional block. The columns are specified by a group of `<loc>` tags and there are three ways to use this tag:

```
<loc type="col" start="<int>" repeat="<int>" priority="<int>"/>
```

This specifies an absolute column assignment. The first column to contain this functional block is specified in start. Every column that satisfies *x = start_x + k*repeat*, where k is any integer, will be composed of this functional block.

```
<loc type="rel" pos="0.5"  priority="<int>"/>
```

This specifies a single column to be composed of this functional block where the column is specified as a fraction of the width.

```
<loc type="fill"  priority="1"/>
```

This is a special specification such that all unspecified columns get assigned this functional block.

For all three `<loc>` tags, the priority attribute is used to resolve collisions when two different functional block is supposed to use the same column. The larger integer specified for priority gets the location.

<timing>*content*</**timing>**

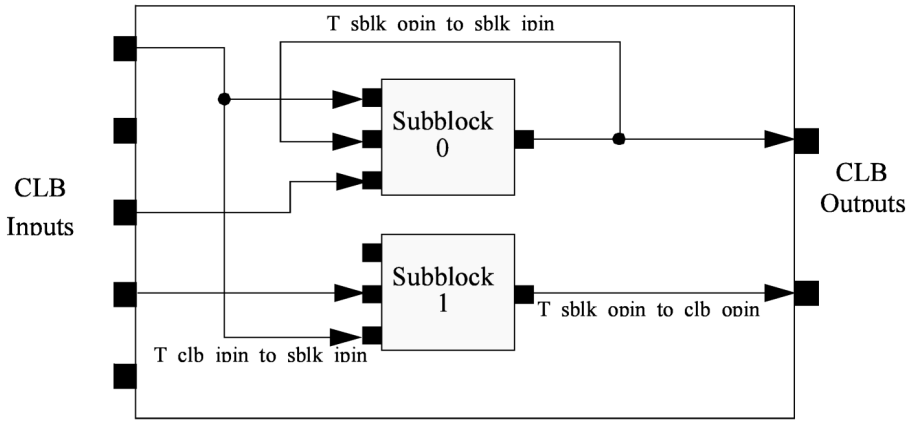
Optional. This timing is specifically for the paths from the functional blocks to the subblocks. The content for this tag specifies timing and is used and must be specified for timing analysis.

<tedge type"{ T_sblk_opin_tosblk_ipin | T_fb_ipin_to_sblk_ipin | T_sblk_opin_to_fb_opin}">*float*</**tedge>**

This tag tedge describes a timing edge for each of the three possibilities:

- T_sblk_opin_to_fb_opin – Delay from the output of a subblock to a clb (logic block) output pin. For architectures without local routing (e.g. the output of a LUT is hardwired to each logic block output), this delay is essentially zero.
- T_fb_ipin_to_sblk_ipin – Delay from an input pin of a clb (logic block) to an input pin of a subblock within that clb. For architectures without local routing (i.e. clb input pins connect directly to some logic element, like a LUT or multiplexer) this delay is essentially zero.
- T_sblk_opin_to_sblk_ipin - Delay from the output of a subblock to the input of another subblock within the same clb. For architectures without local routing (e.g. the output of one subblock is hard- wired to the input of another) this delay is essentially zero.

All three must be included for timing analysis.



<subblocks max_subblocks="*int***" max_subblock_inputs="***int***">***content*</**subblocks>**

Contains the information for the subblocks of a functional block.

Max_subblocks describes how many subblocks there are (in the case of clbs, this is N), and max_subblock_inputs describes the number of inputs per subblocks (in the case of clbs, this is K). For something like a multiplier, we would expect the number of subblocks to be 1 and the

number of input pins to match that of the functional block.

The content of subblocks contains timing information for each subblock. However, the timing information can only be specified such that all subblocks have the same timing characteristics (meaning only one timing can be provided). The content is described in the next table.

The content within subblocks of a functional block is described in the table below. Note that there are three types of timing (one for the chip, one for the paths between functional blocks and subblocks, and one for internal subblock paths). The timing described next is for internal subblock paths between inputs and outputs.

<timing>*content***</timing>**
Optional. This timing is specifically to be embedded in the subblocks content. The content for this tag specifies timing and is used and must be specified for timing analysis. This content consists of T_comb, T_seq_in and T_seq_out as explained below
<T_comb>
The delay from any subblock input to the subblock output when this subblock is used in combinational mode. A subblock is used in combinational mode when the netlist leaves its clock pin OPEN. This describes the timing characteristic for all inputs to all outputs of the subblock. Each input within the subblock will have an associated <trow> (as described below) in which a timing number is provided for the time of the combinational input from this input to the collumn ordered outputs of the subblock. The trows are listed in sequential order from the first input to the last input (row ordered). This represents a matrix that describes the timing characteristics between all inputs and outputs of the subblock.
<T_seq_in>
The delay from any subblock input pin to the FF storage element when this subblock is used in sequential mode. A subblock is used in sequential mode when the netlist hooks its clock pin to some signal. If this subblock was a simple flip flop, for example, then T_seq_in is the setup time. If this subblock corresponds to, say, a LUT feeding into a flip flop, then T_seq_in should be set to the LUT delay plus the setup time. Each entry is a <trow> (as described below) which describes the timing for an output from the first output pin to the last output pin.
<T_seq_out>
The delay from the subblock storage element (FF) to the subblock output pin when this block is used in sequential mode. A subblock is used in sequential mode when the netlist hooks its clock pin to some signal. If this subblock had a flip flop hooked to its output pin, for example, then T_seq_out would be the clock-to-Q delay of the flip flop. Each entry is a <trow> (as described below) which describes the timing for an output from the first output pin to the last output pin.
<trow>*float***</trow>**
A timing container that when nested between: ▪ T_comb – specifies the timing between an input pin and collumn ordered outputs of a subblock ▪ T_seq_in – specifies the timing charactersitc of the sequential input (JASON) ▪ T_seq_out – specifies the timing characteristic of the sequential output (JASON)

One special case in the <typelist> is the input/output pads. They have the following settings.

<io capacity="*int***" t_inpad="***float***" t_outpad="***float***">**content**</io>**
This contains the details for the input and output pads around the periphery of the chip. The following attributes are specified: ▪ capacity is the number of I/Os that are contained per io pad. This means that multiple ios can be contained in one io pad. ▪ t_inpad is the delay through an input pad. ▪ t_outpad is the delay through an output pad.
<fc_in type="{frac|abs|full}">{<int> | <float>}</fc_in>
Content: Sets the number of tracks to which each logic block input pin connects in each channel bordering the pin. The F_c value used is always the minimum of the specified F_c and the channel width, W. It is best to set the type attribute to *full* if you want F_c to always be W. type attribute: The type attribute indicates whether the F_c [12] value should be interpreted as the number of tracks to which each pin connects (*absolute*), or the fraction of tracks in a channel to which each pin connects (*fractional).* Note: type absolute or fractional for Fc_in and Fc_out must be the same. Full disregards whether it is fractional or absolute and can be used in either case.
<fc_out type="{frac|abs|full}">{<int> | <float>}</fc_out>
Content: Sets the number of tracks to which each logic block output pin connects in each channel bordering the pin. Type attribute is the same as described above in Fc_in

6.2.3 Description of the Wire Segments

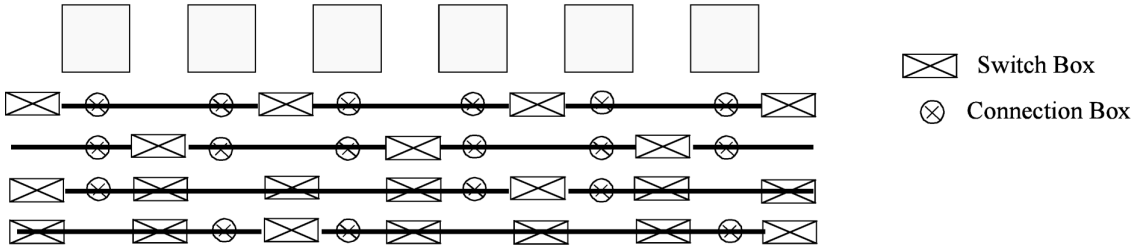
The content within the `<segmentlist>` tag consists of a group of `<segment>` tags. The `<segment>` tag and its contents are described in the table below.

****content****
Describes the properties of a segment ***length****:* Either the number of logic blocks spanned by each segment, or the keyword *longline*. Longline means segments of this type span the entire FPGA array. ***freq****:* The supply of routing tracks composed of this type of segment. VPR automatically determines the percentage of tracks for each segment type by taking the frequency for the type specified and dividing with the sum of all frequencies. It is recommended that the sum of all segment frequencies be in the range 1 to 100. ***Rmetal****:* Resistance per unit length (in terms of logic blocks) of this wiring track, in Ohms. For example, a segment of length 5 with Rmetal = 10 Ohms / logic block would have an end-to-end resistance of 50 Ohms. ***Cmetal****:* Capacitance per unit length (in terms of logic blocks) of this wiring track, in Farads. For example, a segment of length 5 with Cmetal = 2e-14 F / logic block would have a total metal capacitance of 10e-13F. **directionality**: This is either *uni_directional* or *bi_directional* and indicates whether a segment has multiple drive points (*bi_directional*), or a single driver at one end of the wire segment (*uni_directional*). All segments must have the same directionality value. See [15] for a description of uni-directional single-driver wire segments.

Content contains the switch names and the depopulation pattern as described below.

<sb type="pattern">*int list***</sb>**

This tag describes the switch block depopulation (as illustrated in the figure below) for this particular wire segment. For example, the firsth length 6 wire in the figure below has an sb pattern of "1 0 1 0 1 0 1". The second wire has a pattern of "0 1 0 1 0 1 0". A "1" indicates the existance of a switch block and a "0" indicates that there is no switch box at that point. Note that there a 7 entries in the integer list for a length 6 wire. For a length L wire there must be L+1 entries seperated by spaces.



<cb type="pattern">*int list***</cb>**

This tag describes the connection block depopulation (as illustrated by the circles in the figure above) for this particular wire segment. For example, the firsth length 6 wire in the figure below has an sb pattern of "1 1 1 1 1 1". The third wire has a pattern of "1 0 0 1 1 0". A "1" indicates the existance of a connection block and a "0" indicates that there is no connection box at that point. Note that there a 6 entries in the integer list for a length 6 wire. For a length L wire there must be L entries seperated by spaces.

<mux name="*match name*"/>

Option for UNIDIRECTIONAL only. Tag must be included and the "match name" must be the same as the name you give in <switch type="mux" name="…

<wire switch name="*match name*"/>

Option for BIDIRECTIONAL only. Tag must be included and the "match name" must be the same as the name you give in <switch type="buffer" name="… for the switch which represents the wire switch in your architecture.

wire_switch**:* The index of the switch type used by other wiring segments to drive this type of segment. That is, switches going ***to this segment from other pieces of wiring will use this type of switch.

<wire switch name="*match name*"/>

Option for BIDIRECTIONAL only. Tag must be included and the "match name" must be the same as the name you give in <switch type="buffer" name="… for the switch which represents the output pin switch in your architecture.

***opin_switch**:* The index of the switch type used by clb and pad output pins to drive this type of segment.

*NOTE**:** In unidirectional segment mode, there is only a single buffer on the segment. Its type is specified by assigning the same switch index to both wire_switch and opin_switch. VPR will error out if these two are not the same.*

NOTE: The switch used in unidirectional segment mode must be buffered.

6.2.4 Description of the Switch list

The content within the <switchlist> tag consists of a group of <switch> tags. The <switch> tag and its contents are described in the table below.

<switch type="{buffered|mux}" name="*unique name*" R="*float*" Cin=" *float*" Cout=" *float*" Tdel=" *float*" buf_size="*float*" mux_trans_size="*float*"/>
Describes a a type of switch. This statement defines what a certain type of switch is -- segment statements refer to a switch types by their number (the number right after the switch keyword). The various values are: ***name:*** is a unique alphanumeric string which needs to match the segment definition (see above) ***buffered****:* if this switch is a tri-state buffer ***mux:*** if this is a multiplexer ***R****:* resistance of the switch. ***Cin****:* Input capacitance of the switch. ***Cout****:* Output capacitance of the switch. ***Tdel****:* Intrinsic delay through the switch. If this switch was driven by a zero resistance source, and drove a zero capacitance load, its delay would be Tdel + R * Cout. The 'switch' includes both the mux and buffer when in unidirectional mode. ***buf_size:*** [Only for unidirectional and optional] May only be used in unidirectional mode. This is an optional parameter that specifies area of the buffer in minimum-width transistor area units. If not given will be determined from R value. This allows you to use timing models without R's and C's and still be able to measure area. ***mux_trans_size:*** [Only for unidirectional and optional] This parameter must be used if and only if unidirectional segments are used since bidirectional mode switches don't have muxes. The value controls the size of each transistor in the mux, measured in minimum width transistors. The mux is a two-level mux.

6.2.5 An Example Architecture Specification

The listing below is for an FPGA with all channels of the same width, and a clb compatible with that produced by T-VPack with the -no_clustering option. This clb contains 10 4-input LUTs and flip flops; the input pins are listed first, followed by the clb output pin, followed by the clock pin. Notice that the four inputs all have the same pin class, indicating that they are logically equivalent and the router may connect nets to any one of them. The example also includes a heterogeneous block called "leb" in the typelist. It has a similar definition to the clb except more details for timing can be specified for input pin to output pin delay.

```
<!-- VPR Architecture Specification File -->
<!--
Quick XML Primer:
       -> Data is hierarchical and composed of tags (similar to HTML)
       -> All tags must be of the form <foo>content</foo> OR <foo /> with the
          latter form indicating no content. Don't forget the slash at the end.
       -> Inside a start tag you may specify attributes in the form key="value".
          Refer to manual for the valid attributes for each element.
       -> Comments may be included anywhere in the document except inside a tag
          where it's attribute list is defined.
    -> Comments may contain any characters except two dashes.
-->

<architecture>
       <layout auto="1.0"/>
       <!-- fixed size layout example
       <layout width="15" height="15"/>
       -->
       <device>
              <sizing R_minW_nmos="5726.87" R_minW_pmos="15491.7" ipin_mux_trans_size="1"/>
              <timing C_ipin_cblock="1.191e-14" T_ipin_cblock="1.482e-10"/>
```

```
		<area grid_logic_tile_area="100000.0"/>
		<chan_width_distr>
			<io width="1.0"/>
			<x distr="uniform" peak="1"/>
			<y distr="uniform" peak="1"/>
			<!-- Example of different chan width distributions for global routing
			<x distr="gaussian" width="0.5" peak="0.8" xpeak="0.6" dc="0.2"/>
			<y distr="pulse" width="0.5" peak="0.8" xpeak="0.6" dc="0.2"/>
			-->
		</chan_width_distr>
		<switch_block type="wilton" fs="3"/>
	</device>

	<switchlist>
		<!--
			type can be
			name is any unique alphanumeric string
		-->
		<!-- unidir example -->
		<switch type="mux" name="normal" R="94.841" Cin="1.537e-14" Cout="2.194e-13"
		Tdel="6.562e-11" buf_size="16.0" mux_trans_size="1.2"/>

		<!-- bidir example
		<switch type="buffer" name="1" R="94.841" Cin="1.537e-14" Cout="2.194e-13"
		Tdel="6.562e-11"/>
		<switch type="buffer" name="2" R="94.841" Cin="1.537e-14" Cout="2.194e-13"
		Tdel="6.562e-11"/>
		 -->

	</switchlist>

	<segmentlist>
		<!-- unidir example -->
		<segment freq="4" length="1" type="unidir" Rmetal="11.06455" Cmetal="4.72786e-14">
			<mux name="normal" />
			<sb type="pattern">1 1</sb>
			<cb type="pattern"> 1 </cb>
		</segment>
		<segment type="unidir" length="4" freq="1" Rmetal="44.06455" Cmetal="1.72786e-13">
			<mux name="normal" />
			<sb type="pattern">1 0 1 0 1</sb>
			<cb type="pattern"> 1 0 0 1 </cb>
		</segment>

		<!-- bidir example
		<segment length="4" type="bidir" Rmetal="11.06455" Cmetal="4.72786e-14">
			<wire_switch name="1" />
			<opin_switch name="2" />
			<sb type="pattern">11111</sb>
			<cb type="pattern">1111</cb>
		</segment>
		-->

	</segmentlist>

	<typelist>

		<!-- This block defines our IOs.  IOs are are a special type -->
		<io capacity="3" t_inpad="2e-09" t_outpad="1.5e-09">
			<fc_in type="abs">8</fc_in>
			<fc_out type="full" />
		</io>

		<!-- This is a basic CLB block with K=4, N=10, and I=22. The pins are
		logicaly equivalent with one class for input and one for output. -->
		<type name=".clb">
			<subblocks max_subblocks="10" max_subblock_inputs="4">
				<timing>
					<T_comb>
						<!-- matrix row is input pin, column is output pin -
->
						<trow>2e-09</trow>
						<trow>3e-10</trow>
						<trow>4e-10</trow>
						<trow>5e-10</trow>
					</T_comb>
```

```
                        <T_seq_in>
                                <trow>-1e-12</trow>
                        </T_seq_in>
                        <T_seq_out>
                                <trow>5e-10</trow>
                        </T_seq_out>
                </timing>
        </subblocks>

        <fc_in type="frac">1</fc_in>

        <!-- other forms
        <fc_in type="abs">8</fc_in>
        <fc_in type="frac">0.3</fc_in>
        -->

        <fc_out type="full"></fc_out>

        <pinclasses>
                <!-- Logical Equivalence Classes-->
                <class type="in">0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
                20 21</class>
                <class type="out">22 23 24 25 26 27 28 29 30 31</class>
                <class type="global">32</class>
        </pinclasses>

        <pinlocations>
                <!-- Balenced around sides. See LEB type about how offsets work -->
                <loc side="left">0 4 8 12 16 20 24 28 32</loc>
                <loc side="top">1 5 9 13 17 21 25 29</loc>
                <loc side="right">2 6 10 14 18 22 26 30</loc>
                <loc side="bottom">3 7 11 15 19 23 27 31</loc>
        </pinlocations>

        <!-- for priority, bigger is higher priority -->
        <gridlocations>
                <loc type="fill"  priority="1"/>
        </gridlocations>

        <timing>
                <tedge type="T_sblk_opin_to_sblk_ipin">2.5e-10</tedge>
                <tedge type="T_fb_ipin_to_sblk_ipin">3.5e-10</tedge>
                <tedge type="T_sblk_opin_to_fb_opin">4.5e-10</tedge>
        </timing>
</type>

<!-- This is an example definition of a 1x3 LEB with LE -->
<type name=".leb" height="3">
        <subblocks max_subblocks="1" max_subblock_inputs="4"
        max_subblock_outputs="4">
                <timing>
                        <T_comb>
                                <!-- matrix row order is input pin, column is output
                                pin -->
                                <trow>2e-09 2e-09 2e-09 2e-09</trow>
                                <trow>2e-09 2e-09 2e-09 2e-09</trow>
                                <trow>2e-09 2e-09 2e-09 2e-09</trow>
                                <trow>2e-09 2e-09 2e-09 2e-09</trow>
                        </T_comb>
                        <T_seq_in>
                                <trow>-1e-10</trow>
                                <trow>-1e-10</trow>
                                <trow>-1e-10</trow>
                                <trow>-1e-10</trow>
                        </T_seq_in>
                        <T_seq_out>
                                <trow>1e-10</trow>
                                <trow>1e-10</trow>
                                <trow>1e-10</trow>
                                <trow>1e-10</trow>
                        </T_seq_out>
                </timing>
        </subblocks>
        <fc_in type="frac">0.25</fc_in>
        <fc_out type="full" />
        <pinclasses>
```

```
                        <class type="in">0 1 2 3 4</class>
                        <class type="out">5 6 7 8</class>
                        <!—must have clock even if not used by block-->
                        <class type="global">9</class>
                    </pinclasses>

                    <pinlocations>
                        <loc side="left">0 8 </loc>
                        <loc side="left" offset="1">1 9</loc>
                        <loc side="left" offset="2">2</loc>
                        <loc side="top" offset="2">3</loc>
                        <loc side="right">4</loc>
                        <loc side="right" offset="1">5</loc>
                        <loc side="right" offset="2">6</loc>
                        <loc side="bottom">7</loc>
                    </pinlocations>

                    <gridlocations>
                        <loc type="col" start="2" repeat="5" priority="2"/>
                        <loc type="rel" pos="0.5" priority="3"/>
                    </gridlocations>

                    <timing>
                        <tedge type="T_sblk_opin_to_sblk_ipin">2e-9</tedge>
                        <tedge type="T_fb_ipin_to_sblk_ipin">3e-9</tedge>
                        <tedge type="T_sblk_opin_to_fb_opin">4e-9</tedge>
                    </timing>
                </type>

        </typelist>
</architecture>
```

Notice that all the inputs are of the same class, indicating they are all logically equivalent, and all the outputs are of the same class, indicating they are also logically equivalent. This is true of all cluster-based logic blocks, as the local routing within the block provides full connectivity. However, for most logic blocks all the inputs and all the outputs are *not* logically equivalent. For example, consider the logic block in Figure , which consists of a 3-input and gate and a 2-input or gate. In this case, the set {in1, in2, in3} is logically equivalent, and could all be made class 0. Similarly, the set {in4, in5} is logically equivalent, and could be made class 1. Out1 and out2 are obviously not logically equivalent, so each must be a different class, say class 2 and class 3. This may also be the case for heterogeneous blocks where pins are not considered logically equivalent. It is expected that the upstream packed (Tvpack in the case of logic and Odin in the case of multipliers) is aware of this and handles the netlist manipulation accordingly.

6.3 Placement File Format:

The first line of the placement file lists the netlist (.net) and architecture (.arch) files used to create this placement. This information is used to ensure you are warned if you accidentally route this placement with a different architecture or netlist file later. The second line of the file gives the size of the logic block array used by this placement.

All the following lines have the format:

```
block_name    x        y    subblock_number
```

The block name is the name of this block, as given in the input netlist. X and y are the row and column in which the block is placed, respectively. The subblock number is meaningful only for pads. Since we can have more than one pad in a row or column when io_rat is set to be greater than 1 in the architecture file, the subblock number specifies which of the several possible pad locations in row x and column y contains this pad. Note that the first pads

occupied at some (x, y) location are always those with the lowest subblock numbers -- i.e. if only one pad at (x, y) is used, the subblock number of the I/O placed there will be zero. For clbs, the subblock number is always zero.

The placement files output by VPR also include (as a comment) a fifth field: the block number. This is the internal index used by VPR to identify a block -- it may be useful to know this index if you are modifying VPR and trying to debug something.

Figure shows the coordinate system used by VPR via a small 2 x 2 clb FPGA. The number of clbs in the x and y directions are denoted by nx and ny, respectively. Clbs all go in the area with x between 1 and nx and y between 1 and ny, inclusive. All pads either have x equal to 0 or nx + 1 or y equal to 0 or ny + 1.

An example placement file is given below.

```
Netlist file: xor5.net   Architecture file: sample.arch
Array size: 2 x 2 logic blocks

#block name x     y     subblk      block number
#---------- --    --    ------      ------------
a           0     1     0           #0  -- NB: block number is a comment.
b           1     0     0           #1
c           0     2     1           #2
d           1     3     0           #3
e           1     3     1           #4
out:xor5    0     2     0           #5
xor5        1     2     0           #6
[1]         1     1     0           #7
```

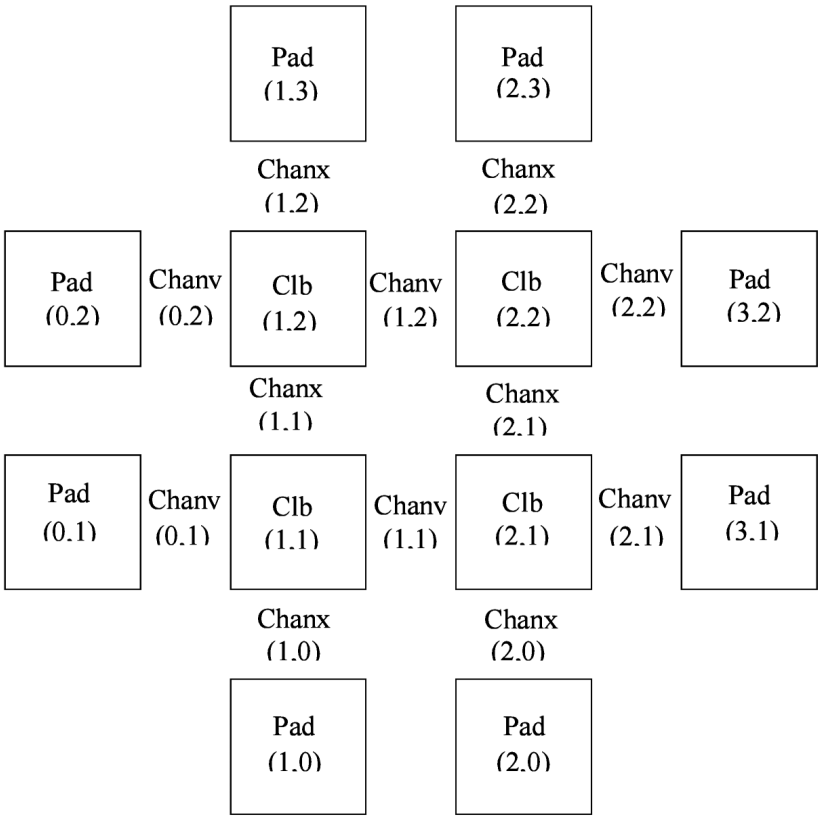


Figure 10: Coordinate system used by VPR.

The blocks in a placement file can be listed in any order.

6.4 Routing File Format

The first line of the routing file gives the array size, nx x ny. The remainder of the routing file lists the global or the detailed routing for each net, one by one. Each routing begins with the word net, followed by the net index used internally by VPR to identify the net and, in brackets, the name of the net given in the netlist file. The following lines define the routing of the net. Each begins with a keyword that identifies a type of routing segment. The possible keywords are SOURCE (the source of a certain output pin class), SINK (the sink of a certain input pin class), OPIN (output pin), IPIN (input pin), CHANX (horizontal channel), and CHANY (vertical channel). Each routing begins on a SOURCE and ends on a SINK. In brackets after the keyword is the (x, y) location of this routing resource. Finally, the pad number (if the SOURCE, SINK, IPIN or OPIN was on an I/O pad), pin number (if the IPIN or OPIN was on a clb), class number (if the SOURCE or SINK was on a clb) or track number (for CHANX or CHANY) is listed -- whichever one is appropriate. The meaning of these numbers should be fairly obvious in each case. If we are attaching to a pad, the pad number given for a resource is the subblock number defining to which pad at location (x, y) we are attached. See Figure for a diagram of the coordinate system used by VPR. In a horizontal channel (CHANX) track 0 is the bottommost track, while in a vertical channel (CHANY) track 0 is the leftmost track. Note that if only global routing was performed the track number for each of the CHANX and CHANY resources listed in the routing will be 0, as global routing does not assign tracks to the various nets.

For an N-pin net, we need N-1 distinct wiring “paths” to connect all the pins. The first wiring path will always go from a SOURCE to a SINK. The routing segment listed immediately after the SINK is the part of the existing routing to which the new path attaches. *It is important to realize that the first pin after a SINK is the connection into the already specified routing tree; when computing routing statistics be sure that you do not count the same segment several times by ignoring this fact.* An example routing for one net is listed below.

```
Net 5 (xor5)

SOURCE (1,2)  Class: 1        # Source for pins of class 1.
 OPIN (1,2)  Pin: 4
 CHANX (1,1)  Track: 1
 CHANX (2,1)  Track: 1
 IPIN (2,2)  Pin: 0
 SINK (2,2)  Class: 0        # Sink for pins of class 0 on a clb.
 CHANX (1,1)  Track: 1        # Note:  Connection to existing routing!
 CHANY (1,2)  Track: 1
 CHANX (2,2)  Track: 1
 CHANX (1,2)  Track: 1
 IPIN (1,3)  Pad: 1
 SINK (1,3)  Pad: 1      # This sink is an output pad at (1,3), subblock 1.
```

Nets which are specified to be global in the netlist file (generally clocks) are not routed. Instead, a list of the blocks (name and internal index) which this net must connect is printed out. The location of each block and the class of the pin to which the net must connect at each block is also printed. For clbs, the class is simply whatever class was specified for that pin in the architecture input file. For pads the pinclass is always -1; since pads do not have logically-equivalent pins, pin classes are not needed. An example listing for a global net is given below.

```
Net 146 (pclk): global net connecting:
Block pclk (#146) at (1, 0), pinclass -1.
Block pksi_17_ (#431) at (3, 26), pinclass 2.
```

```
Block pksi_185_ (#432) at (5, 48), pinclass 2.
Block n_n2879 (#433) at (49, 23), pinclass 2.
```

7. Debugging Aids

After parsing the netlist and architecture files, VPR dumps out an image of its internal data structures into net.echo and arch.echo. These files can be examined to be sure that VPR is parsing the input files as you expect. The critical_path.echo file lists details about the critical path of a circuit, and is very useful for determining why your circuit is so fast or so slow. Various other data structures can be output if you uncomment the calls to the output routines; search the code for *echo* to see the various data that can be dumped.

If the preprocessor flag DEBUG is defined in vpr_types.h, some additional sanity checks are performed during a run. I normally leave DEBUG on all the time, as it only slows execution by 1 to 2%. The major sanity checks are always enabled, regardless of the state of DEBUG. Finally, if VERBOSE is set in vpr_types.h, a great deal of intermediate data will be printed to the screen as VPR runs. If you set verbose, you may want to redirect screen output to a file.

The initial and final placement costs provide useful numbers for regression testing the netlist parsers and the placer, respectively. I generate and print out a routing serial number to allow easy regression testing of the router.

Finally, if you need to route an FPGA whose routing architecture cannot be described in VPR's architecture description file, don't despair! The router, graphics, sanity checker, and statistics routines all work only with a graph that defines all the available routing resources in the FPGA and the permissible connections between them. If you change the routines that build this graph (in rr_graph*.c) so that they create a graph describing your FPGA, you should be able to route your FPGA. If you want to read a text file describing the entire routing resource graph, call the dump_rr_graph subroutine.

8. References

[1] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton. "Benchmarking method and designs targeting logic synthesis for FPGAs", Proc. IWLS '07, pp. 230-237.

[2] S. Cho, S. Chatterjee, A. Mishchenko, and R. Brayton, "Efficient FPGA mapping using priority cuts". (Poster.) Proc. FPGA '07.

[3] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, 1999.

[4] V. Betz, "Architecture and CAD for the Speed and Area Optimization of FPGAs," *Ph.D. Dissertation*, University of Toronto, 1998.

[5] V. Betz and J. Rose, "Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size," *CICC,* 1997, pp. 551 - 554.

[6] A. Marquardt, V. Betz and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," *ACM/SIGDA Int. Symp. on FPGAs*, 1999, pp. 37 - 46.

[7] V. Betz and J. Rose, "Directional Bias and Non-Uniformity in FPGA Global Routing Architectures," *ICCAD*, 1996, pp. 652 - 659.

[8] V. Betz and J. Rose, "On Biased and Non-Uniform Global Routing Architectures and CAD Tools for FPGAs," *CSRI Technical Report #358*, Department of Electrical and Computer Engineering, University of Toronto, 1996. *(Available for download from http://www.eecg.toronto.edu/~vaughn/ papers/techrep.ps.Z).*

[9] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," *Seventh International Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213 - 222.

[10] A. Marquardt, V. Betz and J. Rose, "Timing-Driven Placement for FPGAs," *ACM/SIGDA Int. Symp. on FPGAs*, 2000, pp. 203 - 213.

[11] V. Betz and J. Rose, "Automatic Generation of FPGA Routing Architectures from High-Level Descriptions," *ACM/SIGDA Int. Symp. on FPGAs*, 2000, pp. 175 - 184.

[12] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[13] S. Wilton, "Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories," *Ph.D. Dissertation*, University of Toronto, 1997. *(Available for download from http://www.ece.ubc.ca/~stevew/publications.html).*

[14] Y. W. Chang, D. F. Wong, and C. K. Wong, "Universal Switch Modules for FPGA Design," *ACM Trans. on Design Automation of Electronic Systems*, Jan. 1996, pp. 80 - 101.

[15] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Direction and Single-Driver Wires in FPGA Interconnect," *International Conference on Field-Programmable Technology,* 2004, pp. 41-48

[16] A Verilog RTL Synthesis Tool For Heterogeneous FPGAs. Peter Jamieson and Jonathan Rose. *15th International Conference on Field Programmable Logic and Applications,*

August, 2005.